UDP protokol

Prilikom osmišljavanja aplikacije koja će koristiti internet protokol u osnovi, postavlja se pitanje da li će komunikacija na mreži koristiti uređeni i pouzdani niz paketa ili ne i, u zavisnosti od toga, izbora između UDP i TCP protokola. Iako je TCP protokol popularniji, UDP protokol nam, zahvaljujući svojoj jednostavnosti, daje brži i pregledniji uvid u ponašanje paketa prilikom komunikacije, što će nam biti potrebno u nastavnoj jedinici o TCP protokolu. Takođe nam omogućava i brže upoznavanje sa Python mrežnim okruženjem.

Umesto razvoja sopstvenih aplikativnih okruženja u Pythonu za rad sa soketima, koristićemo ugrađena rešenja. Naime, Python nam omogućava rad sa soketima uz korišćenje ugrađene biblioteke socket, koja pruža objektno orijentisano okruženje za rad sa računarskom mrežom. Dakle, ova biblioteka nam pruža pristup liniji komunikacije – kao što su UDP ili TCP portovi, koje možemo koristiti kao objekte u Pythonu. Nakon definisanja jednog takvog objekta, potrebno ga je i povezati sa željenim soketom. Soketi u Pythonu su dosta slični fajl deskriptorima, ali se, umesto na disku, nalaze na mreži i podržavaju operacije kao što su .read().i .write(), što nam omogućava lako pisanje u soket i čitanje iz njega. Upoznavanje sa osnovnim principima UDP protokola ćemo pokazati na primeru komunikacije klijenta sa serverom čiji tok komunikacije izgleda ovako:

- Klijent šalje tip podatka string zadat u promenljivoj 'input_s'.
- Server prima ovaj podatak i vrši deljenje tog stringa na osnovu karaktera zapete (,) kao graničnika.
- Ovako izmenjen string server šalje nazad klijentu.
- Klijent prima izmenjen string i ispisuje ga na ekranu.

Fajl za UDP server će nam se zvati udp_local_server.py, dok će se fajl za UDP klijent zvati udp_local_client.py. I server i klijent će se pokretati sa lokalnog servera (localhost) i pokretaće se iz komandnog prozora (Command Prompt) komandama python udp_local_server.py i python udp_local_client.py.

UDP klijent

Pre kreiranja servera i klijenta, potrebno je najpre uvesti socket bibiloteku komandom import socket, koja je zadužena za osnove mrežne komunikacije u Pythonu i pomoću koje možemo kreirati sokete. Nakon toga ćemo definisati port preko kojeg će se odvijati komunikacija linijom: server_port = 21060. U ovom trenutku je sve spremno za kreiranje soket objekta. To činimo linijom:

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Dakle, definisali smo promenljivu client_socket kao objekat tipa soket. Prosleđeni parametri ovog soketa su:

- AF_INET – Ovim parametrom označavamo verziju internet protokola preko kojeg će se komunikacija odvijati. U ovom slučaju je to IPv4.
- SOCK_DGRAM – Ovim parametrom definišemo kog tipa će soket biti. U ovom slučaju, pošto je reč o UDP protokolu, tip soketa je UDP datagram (SOCK_DGRAM).

U ovom koraku tek kreiramo soket objekat; još uvek mu ne prosleđujemo adresu ka kojoj ga treba poslati, niti port ka kojem treba da pristigne. To ćemo učiniti pošto najpre definišemo poruku koju ćemo poslati: input_s = "Hello, server!". Sledećom naredbom:

```
client_socket.sendto(bytes(input_s, encoding='utf8'), ('127.0.0.1', server_port))
```

koristimo metodu soket objekta – sendto, kojom šaljemo poruku serveru, a kojoj zadajemo sledeće parametre:

- bytes objekat – poruka koju želimo da pošaljemo tipa bajt, u UTF-8 kodnom rasporedu;
- n-torka od dva elementa, koja sadrži adresu na koju šaljemo, kao i port procesa na mašini koja se nalazi iza te adrese. U našem slučaju, adresa je 127.0.0.1. Ova adresa je specifična i predstavlja localhost – lokalni web server. Na svakoj mašini, ova adresa je rezervisana upravo za lokalni server. Takođe, njoj se može pristupiti samo iz lokalne mreže (mreže na nivou samog uređaja), koristeći 127.0.0.1 adresu u browseru ili jednostavno otkucavši localhost.

U ovom trenutku u kodu, poslali smo paket koji sadrži poruku, adresu i port gde poruka treba da stigne. Nakon obrade servera, kada nam stigne povratni paket, taj podatak ćemo prihvatiti linijom:

```
input_s_modified, address = client_socket.recvfrom(65535)
```

Ovom naredbom učitavamo sve što nam stiže od servera. Tačnije, u promenljivoj address su nam adresa i port servera, a u promenljivoj input_s_modified se nalazi odgovor servera. Metoda recvfrom (skraćeno od: receive from) prihvata dva parametra, od kojih je u ovom slučaju samo jedan iskorišćen:

- buffer – bafer predstavlja pomoćnu memoriju i u ovom slučaju ta memorija se koristi za smeštanje odgovora servera. Vrednost koju smo prosledili (65535) nije slučajna i predstavlja veličinu bafera izraženu u bajtovima. Takođe, ta cifra je po specifikaciji protokola teorijski limit veličine poruke koja se može preneti UDP-om. Ako pokušamo da pošaljemo poruku veću od limita, doći će do greške OSError: [WinError 10040], jer pokušavamo slanje poruke veće od limita i u tom slučaju poruku pre slanja treba podeliti;
- flags (indikatore stanja) – predstavlja na koji način će se poruka primiti; vrednost tog parametra se prosleđuje kao tip int. Više o ovom parametru možete saznati na ovoj stranici.

Pored metode recvfrom, postoji i metoda recv, koja prima samo jedan ulazni parametar, odnosno vrednost bafera koja je tipa int. Razlika između recv i recvfrom metoda je u povratnoj vrednosti: recv metoda vraća vrednost kao string, a recvfrom vraća n-torku koja se sastoji od dva elementa, gde je prvi element povratni odgovor servera tipa string, a drugi element je adresa servera sa koje je stigao odgovor.

Nakon što smo primili poruku, možemo je dalje procesuirati. Iako će Python sam zatvoriti konekciju kada se dođe do kraja programa, dobra je praksa da soket ipak mi ručno zatvorimo metodom .close(). Izgled klijentskog dela UDP-a se nalazi u fajlu udp_local_client.py, koji na kraju izgleda ovako:

udp_local_client.py

```
import socket
server_port = 21060
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
input_s = 'Hello, server!'
client_socket.sendto(bytes(input_s, encoding='utf8'), ('127.0.0.1',
server_port))
input_s_modified, address = client_socket.recvfrom(65535)
print ('[CLIENT] Response from server {}, is: "{}"'.format(address,
str(input_s_modified.decode('utf8'))))
client_socket.close()
```

Napomena

Prilikom slanja poruke ka serveru šaljemo tip bytes; takođe, prilikom prijema odgovora dobijamo bytes tip podatka. Za pretvaranje bytes tipa u string i nama čitljiv format, koristimo metodu decode(), kojoj prosleđujemo kodni raspored znakova. Najpoznatiji je UTF-8 način kodiranja. Na taj način dobijamo str tip objekta. A ako želimo da pretvorimo str objekat u bytes, koristimo string metodu encode(), kojoj takođe prosleđujemo kodni raspored znakova. Više o listi kodnih rasporeda znakova (character encoding) može se pročitati ovde.

Pitanje

Za slanje poruke pri UDP protokolu, koristimo metodu:

- write()
- send()
- **sendto()**

Objašnjenje:

Tačan odgovor je metoda sendto(), koja za argumente prima poruku tipa bytes i n-torku sa dva elementa – adresa i port, a koja se koristi u radu sa UDP protokolom.

UDP Server

Prvih par početnih linija kod implementacija UDP servera su iste kao i kod klijenta. U oba slučaja uvozimo socket biblioteku, preciziramo broj porta na kojem će server slušati, odnosno, određujemo na koji će port klijent slati poruku (portovi koji se zadaju na serverskoj i klijentskoj strani moraju biti isti), kao i tip soketa koji kreiramo (u oba slučaja je to datagram soket preko IPv4 protokola):

```
import socket

server_port = 21060

server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Prva naredna razlika u odnosu na kod klijenta je u soketskoj metodi .bind(), kojom našem, lokalnom serveru, tačnije procesu naše aplikacije, dodeljujemo broj porta na kojem će osluškivati.

```
server_socket.bind(('127.0.0.1', server_port))
```

Nakon korišćenja bind metode, ispisaćemo komandom print da je kreiran soket, a komandom getsockname() možemo ispisati adresu i port soketa.

Kako bi server u pravom momentu prepoznao primanje poruke, mora u svakom trenutku da proverava da li je poruka stigla. Ovaj problem, između ostalog, možemo rešiti i koristeći običnu while beskonačnu petlju. Dakle, u svakoj iteraciji while petlje će se proveravati da li je pristigla nova poruka. Ovakvo proveravanje se čini na identičan način kao i kod klijenta:

```
message, address = server_socket.recvfrom(65535)
```

Kao i kod klijenta, i u ovom slučaju dobijamo poruku i adresu koristeći .recvfrom() metodu. Za razliku od klijenta, gde promenljiva address ne igra bitnu ulogu u povratnoj informaciji, serveru je ova promenljiva i te kako bitna kako bi iz nje izvukao adresu i broj porta na koji treba poslati odgovor.

Treba napomenuti da će ova komanda sama po sebi zaustaviti program na ovoj liniji i čekati poruku, ali bez beskonačne petlje; nakon pristizanja poruke, skript bi se završio i više ne bismo bili u mogućnosti da osluškujemo za nove poruke.

Nakon ove linije dolazimo i do glavne logike skripta, tačnije funkcionalnosti koju naš UDP server treba da implementira – da deli pristiglu tekstualnu poruku pomoću graničnika (,) i da vraća brojčanu vrednost – koliko je delova nastalo tom podelom. Ovaj problem se rešava uz pomoć string .split() metode. Važno je napomenuti da je promenljiva message koja pristigne sa mreže pomoću .recvfrom() metode tipa bajt i da je treba prebaciti u string objekat kako bi se mogla koristiti .split() metoda.

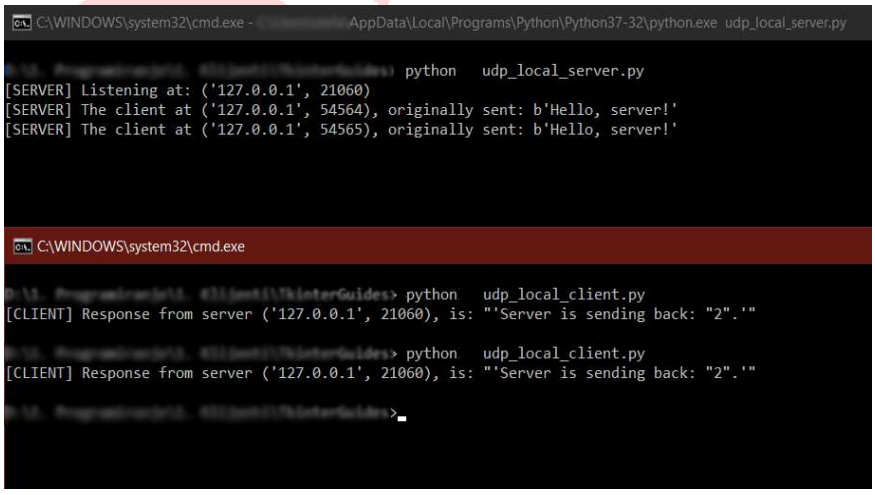
Nakon ovog procesiranja, u odgovor klijentu se smeštaju sama poruka, adresa i port. Način slanja poruke od servera ka klijentu je identičan kao i obrnuto – korišćenjem .sendto() metode kojoj zadajemo poruku u obliku bajt objekta zajedno sa adresom i portom. Kod ovog UDP servera to se nalazi u udp_local_server.py fajlu:

udp_local_server.py

```
import socket
server_port = 21060
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind(('127.0.0.1', server_port))
print ('[SERVER] Listening at:
{}'.format(server_socket.getsockname()))
while True:
    message, address = server_socket.recvfrom(65535)
    modified_message = str(len(str(message).split(",")))
    print('[SERVER] The client at {}, originally sent: {}'.format(address,
repr(message)))
    server_socket.sendto(bytes('Server is \
    sending back: "{}".'.format(modified_message), encoding='utf8'),address)
server_socket.close()
```

Pokretanje i analiza rezultata

Kako bismo proverili funkcionalnost klijenta i servera, moramo pokrenuti oba fajla istovremeno. Ovo ćemo najlakše uraditi koristeći Command Prompt (komandnu liniju). Dakle, prvo otvoriti jedan prozor komandne linije i u njemu pokrenuti udp_local_server.py, a potom pokrenuti i drugi prozor komandne linije i u njemu pokrenut udp_local_client.py. Pre pokretanja ovih komandi je potrebno pozicionirati se u direktorijum gde se ove skripte nalaze (koristeći komandu komandne linije cd). Jedan takav rezultat pokretanja izgleda ovako:



Slika 4.1. Prikaz pokretanja klijenta i servera kroz komandnu liniju

U gornjem prozoru komandne linije je pokrenut UDP server, a u donjem UDP klijent. Odmah po pokretanju servera, ispisuje se string koji kaže da je server spreman za primanje podataka na datoj adresi i portu. U tom trenutku pokrećemo klijentski fajl. Nakon pokretanja klijenta, serveru se šalje poruka „Hello, server!”, na kojoj server primenjuje .split() metodu, broji koliko je elemenata nastalo tom podelom i rezultat šalje nazad klijentu .sendto() metodom, koju klijent prima i taj rezultat ispisuje na ekran.

Rezime

- Za rad sa soketima koristimo ugrađenu Python biblioteku socket.
- Metodom socket() kreiramo soket objekat i njoj prosleđujemo dva argumenta: AF_INET – verzija internet protokola i SOCK_DGRAM – tip soketa (za UDP protokol to je datagram).
- Za konkretno slanje poruke koristimo metodu sendto(), kojoj prosleđujemo dva parametra – bytes tip poruke i n-torku sa dva elementa: adresa i port.
- Za prijem poruke od druge strane koristimo metodu recvfrom().
- Potrebno je da klijent i server koriste istu adresu i port prilikom komunikacije.